Week-3:

TASK1:

Aim: To find the lexicographically smallest and largest substrings of length k from a given string**.**

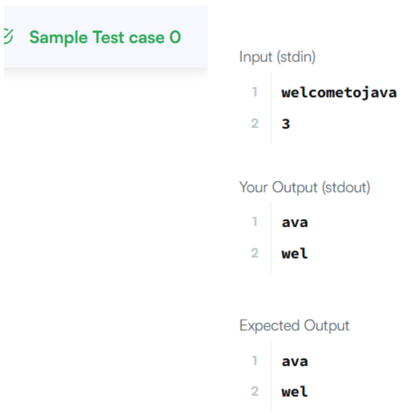
Algorithm:

1. Start.
2. Read the input string s and integer k.
3. Initialize smallest and largest with the first substring of length k.
4. Loop from index 1 to (length of string – k):
5. Extract substring sub of length k.
6. Compare sub with smallest:
7. If sub is smaller, update smallest.
8. Compare sub with largest:
9. If sub is larger, update largest.
10. After checking all substrings, return smallest and largest.
11. Print the result.
12. Stop.

Program:

```
import java.util.Scanner;

public class Solution {

    public static String getSmallestAndLargest(String s, int k) {

        String smallest = s.substring(0, k);

        String largest = s.substring(0, k);

        for (int i = 1; i <= s.length() - k; i++) {

            String sub = s.substring(i, i + k);

            if (sub.compareTo(smallest) < 0)

                smallest = sub;

            if (sub.compareTo(largest) > 0)

                largest = sub;

        }
        return smallest + "\n" + largest;
    }
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.next();
        int k = scan.nextInt();
        scan.close();
        System.out.println(getSmallestAndLargest(s, k));
    }
}
```

Output:



Sample Test case 0

Input (stdin)

```
welcometojava
3
```

Your Output (stdout)

```
ava
wel
```

Expected Output

```
ava
wel
```

Result: The program successfully identifies and displays the smallest and largest substrings of length k from the given string using lexicographical comparison.

Task-2:

Aim: To sort Player objects in descending order based on score and, when scores are equal, in ascending alphabetical order based on name.
This ensures proper ranking with clear tie-breaking using player names.

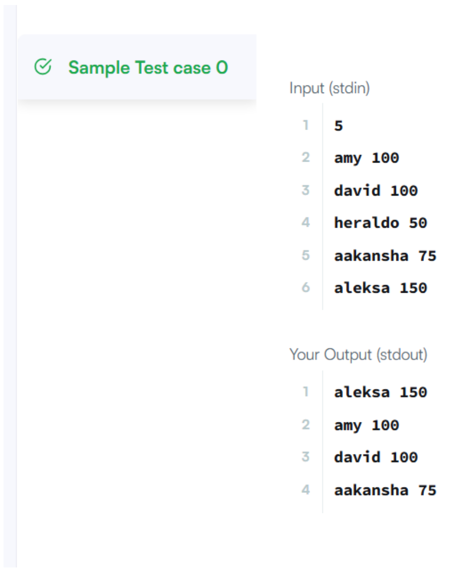
Algorithm:

1. Start.
2. Receive two Player objects: a and b.
3. Compare their scores:
4. If a.score is not equal to b.score, return (b.score – a.score) to sort in descending order.
5. If scores are equal:
6. Compare a.name and b.name using compareTo() to sort alphabetically (ascending).
7. Return the comparison result.
8. Stop.

Program:

```
class Checker implements Comparator<Player> {
 public int compare(Player a, Player b){
        if(a.score != b.score){
            return b.score - a.score;
        }
        return a.name.compareTo(b.name);
    }
```

Output:


Sample Test case 0

Input (stdin)

```
5
amy 100
david 100
heraldo 50
aakansha 75
aleksa 150
```

Your Output (stdout)

```
aleksa 150
amy 100
david 100
aakansha 75
```

Result: Thus, to sort Player objects in descending order based on score and, when scores are equal, in ascending alphabetical order based on name was executed successfully.

Task3:

Aim: To sort a list of students based on CGPA in descending order, first name in ascending alphabetical order, and ID in ascending order when previous values are equal.

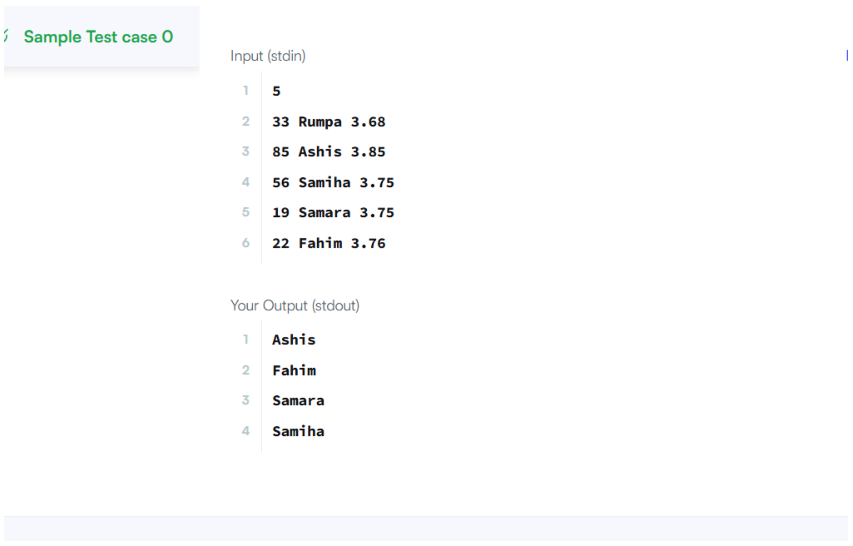
Algorithm:

1. Start the program.
2. Read the number of test cases (students).
3. For each student:
    a. Read ID, first name, and CGPA.
    b. Create a Student object and add it to the list.
4. Use Collections.sort() with a custom Comparator to sort students by:
    a. CGPA (descending order).
    b. First name (ascending order) if CGPA is same.
    c. ID (ascending order) if both CGPA and name are same.
5. Display the sorted student names.
6. Stop.

Program:

```
import java.util.*;

class Student {

   int id;

   String fname;

   double cgpa;

   public Student(int id, String fname, double cgpa) {

      this.id = id;

      this.fname = fname;

      this.cgpa = cgpa;

   }

   public int getId() {

      return id;

   }

   public String getFname() {
        return fname;
    }
    public double getCgpa() {
        return cgpa;
    }
}
class Solution {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int testCases = Integer.parseInt(in.nextLine());
        List<Student> studentList = new ArrayList<Student>();
        while(testCases > 0){
            int id = in.nextInt();
            String fname = in.next();
            double cgpa = in.nextDouble();
            Student st = new Student(id, fname, cgpa);
            studentList.add(st);
            testCases--;
        }
        Collections.sort(studentList, new Comparator<Student>() {
            public int compare(Student s1, Student s2) {
                if(s1.cgpa != s2.cgpa){
                    return Double.compare(s2.cgpa, s1.cgpa);
                }
                if(!s1.fname.equals(s2.fname)){
                    return s1.fname.compareTo(s2.fname);
                }
            return s1.id - s2.id;
             }
          });
          for(Student st: studentList){
             System.out.println(st.getFname());
          }
       }
    }
```

Output:



Sample Test case 0

Input (stdin)

```
5
33 Rumpa 3.68
85 Ashis 3.85
56 Samiha 3.75
19 Samara 3.75
22 Fahim 3.76
```

Your Output (stdout)

```
Ashis
Fahim
Samara
Samiha
```

Result: Thus ,the program successfully sorts all student records according to the given priority rules and prints the students' first names in the correct ranked order was executed successfully.

Task4:

Aim: To sort people's names based on their heights in descending order, so that the tallest person appears first.

Algorithm:

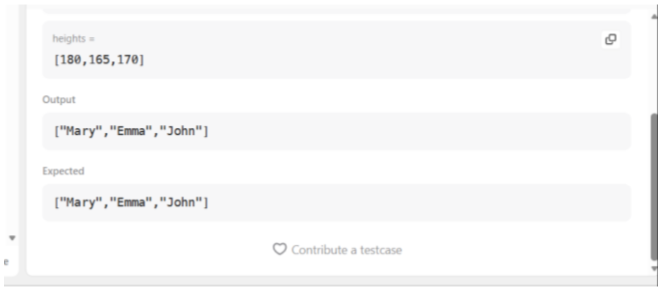
1. Start.
2. Accept two arrays: names[] and heights[].
3. Find the length of the heights array and store it in n.
4. Use Bubble Sort to arrange heights in descending order:
    - Compare each height with the remaining heights.
    - If the current height is smaller, swap both heights.
    - At the same time, swap the corresponding names to maintain mapping.
5. Repeat until all elements are sorted.
6. Return the sorted names array.
7. Stop.

Program:

```
class Solution {
   public String[] sortPeople(String[] names, int[] heights) {
      int n = heights.length;
      for(int i = 0; i < n - 1; i++){
         for(int j = i + 1; j < n; j++){
            if(heights[i] < heights[j]){
               int tempH = heights[i];
               heights[i] = heights[j];
               heights[j] = tempH;
               String tempN = names[i];
               names[i] = names[j];
               names[j] = tempN;
```

```
                }
            }
        }
        return names;
    }
}
```

Output:



```
heights =
[180,165,170]
Output
["Mary","Emma","John"]
Expected
["Mary","Emma","John"]
```

Result: The program successfully sorts the people according to their heights in descending order and returns the names in the correct order, with the tallest person listed first.

Task 6:

Aim: To find the Nth prime number using a Java program.

Algorithm:

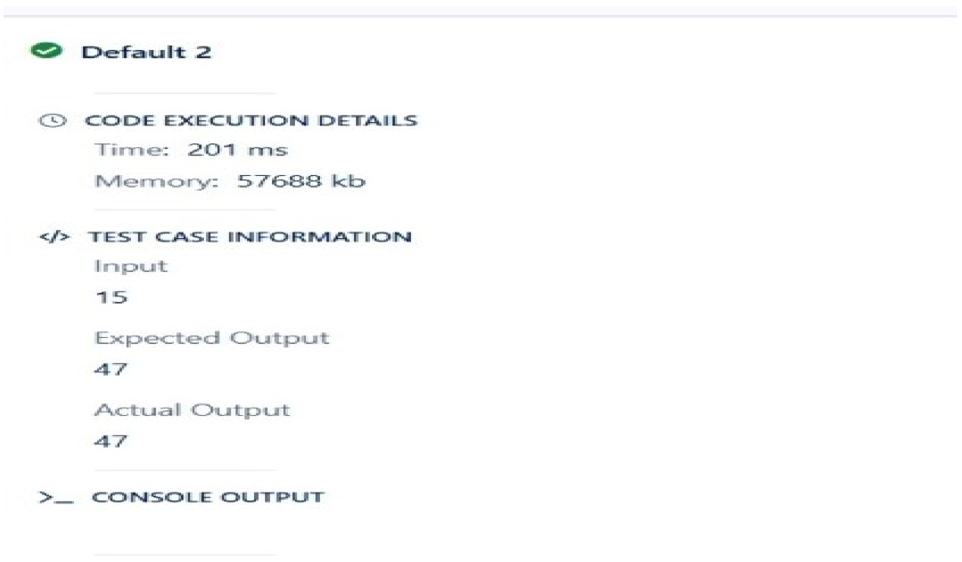
1. Start with count = 0 and num = 1.
2. Increment num by 1.
3. Check whether num is a prime number.
4. If num is prime, increment count.
5. Repeat steps 2–4 until count equals the given input (input1).
6. Return num as the Nth prime number.

Program:

```
class UserMainCode

{

   public int NthPrime(int input1){


      int count = 0;

      int num = 1;


      while(count < input1){

         num++;


         if(isPrime(num)){

            count++;

         }

      }


      return num;

   }
```

```
    private boolean isPrime(int n){

        if(n < 2) return false;

        for(int i = 2; i * i <= n; i++){
            if(n % i == 0)
                return false;
        }

        return true;
    }
}
```

Output:



```
Default 2

CODE EXECUTION DETAILS
  Time:  201 ms
  Memory:  57688 kb

TEST CASE INFORMATION
  Input
  15
  Expected Output
  47
  Actual Output
  47

>_ CONSOLE OUTPUT
```

Result: The program successfully returns the Nth prime number**.**

Task 8:

Aim: To find the lexicographically smallest and largest substrings of length k from a given string.

Algorithm:

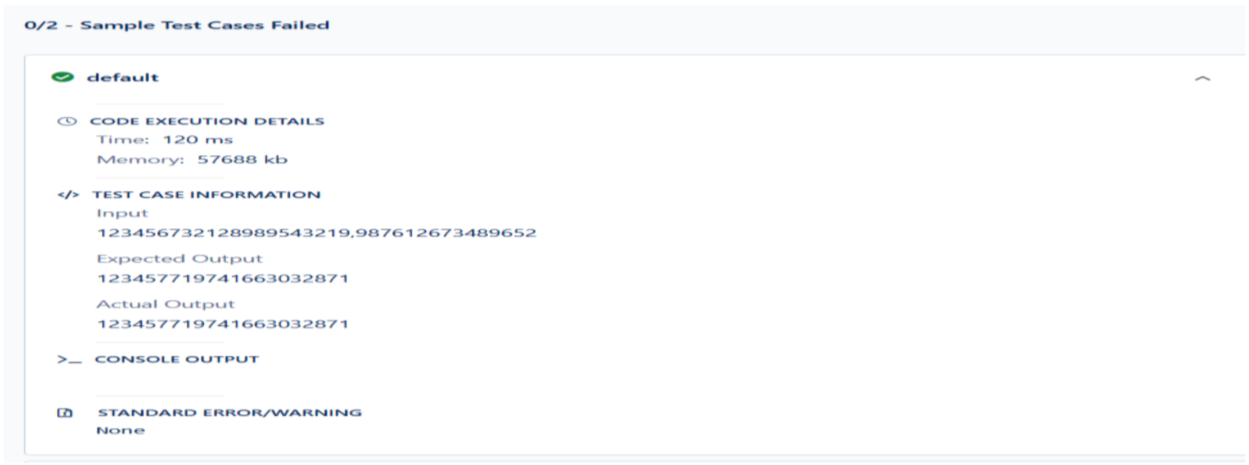
1. Take the first substring of length k as both smallest and largest.
2. Loop from index 1 to length of string − k.
3. Extract each substring of length k.
4. Compare the substring with smallest and largest using compareTo().
5. Update smallest if substring is smaller.
6. Update largest if substring is larger.
7. Print the smallest and largest substrings.

Program:

```
import java.util.Scanner;

public class Solution {

 public static String getSmallestAndLargest(String s, int k) {

         String smallest = s.substring(0, k);

       String largest = s.substring(0, k);



       for (int i = 1; i <= s.length() - k; i++) {

           String sub = s.substring(i, i + k);



           if (sub.compareTo(smallest) < 0)

               smallest = sub;



           if (sub.compareTo(largest) > 0)

               largest = sub;

       }
```

```
        return smallest + "\n" + largest;
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String s = scan.next();
        int k = scan.nextInt();
        scan.close();

        System.out.println(getSmallestAndLargest(s, k));
    }
}
```

Output:



```
0/2 - Sample Test Cases Failed

default

CODE EXECUTION DETAILS
  Time:  120 ms
  Memory:  57688 kb

</> TEST CASE INFORMATION
  Input
  123456732128989543219,987612673489652
  Expected Output
  123457719741663032871
  Actual Output
  123457719741663032871

>_ CONSOLE OUTPUT

STANDARD ERROR/WARNING
  None
```

Result: The programs was successfully find the lexicographically smallest and largest substrings of length k from a given string.

Task 9:

Aim: To compute a result by performing **addition and subtraction on numbers from N to 1** based on the selected option.

Algorithm:

1. Read the value of N and the option opt.
2. Initialize result to 0 and a boolean variable add to control addition/subtraction.
3. Loop from N down to 1.
4. If opt == 1:
5. If opt == 2:
6. Update the result in each iteration.
7. Return the final computed value.

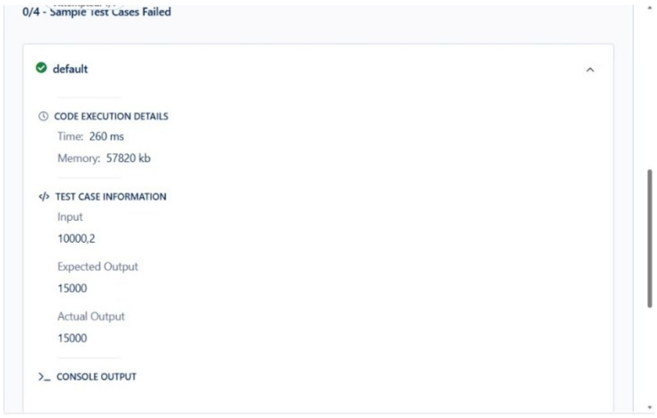
Program:

```
class UserMainCode

{

   public int AddSub(int input1, int input2){



      int N = input1;

      int opt = input2;



      int result = 0;

      boolean add = true;



      if(opt == 1){

         add = true;   // start with +

      }else{

         add = true;   // opt2 also starts with +

      }
```

```
for(int i = N; i >= 1; i--){

    if(opt == 1){
        // Pattern: + - + - ...
        if(add)
            result += i;
        else
            result -= i;

        add = !add;
    }
    else{
        // Pattern: + + - + - ...
        if(i == N || i == N-1){
            result += i;
        }
        else{
            if(add)
                result -= i;
            else
                result += i;

            add = !add;
```

```
                }
            }
        }

        return result;
    }
}
```

Output:



0/4 - Sample Test Cases Failed

default

CODE EXECUTION DETAILS

Time: 260 ms

Memory: 57820 kb

TEST CASE INFORMATION

Input

10000,2

Expected Output

15000

Actual Output

15000

CONSOLE OUTPUT

Result: The program successfully calculates the final value by applying the specified addition and subtraction pattern to the numbers from N to 1 based on the given option.